Sanitizers

a new generation of bug finding tools

Linux Plumbers, Nov 4, 2016, Santa Fe

Dmitry Vyukov, dvyukov@, Google

Agenda

- **AddressSanitizer (KASAN)**: use-after-free, out-of-bounds on heap/stack/globals
- **MemorySanitizer (KMSAN)**: uses of uninit data
- **ThreadSanitizer (KTSAN)**: data races
- Complaints, pain points, and wishes

Our Team

Dynamic Testing Tools team at Google:

- **ASAN**: also container overflows, initialization order bugs, use-after-return, use-after-scope, intra-object-overflows
- **TSAN**: also deadlocks, async-unsafe code, races on fd
- **MSAN**: uses of uninit data
- **UBSAN**: other simple UBs
- **LSAN**: heap leaks
- **CFI, SafeStack**: hardening of prod code
- **libFuzzer**: coverage-guided fuzzer

All in clang, most in gcc.

Why Compiler-based Tools?

- Simple
- Arch-independent
- Fast
- Flexible

Main Principles

- Fast, low memory overhead
- Zero tolerance for false positives
- Easy to use, work out of the box, informative reports

Usage

- Continuous testing
- System testing
- Fuzzing
- Debugging

KASAN

use-after-free, out-of-bounds

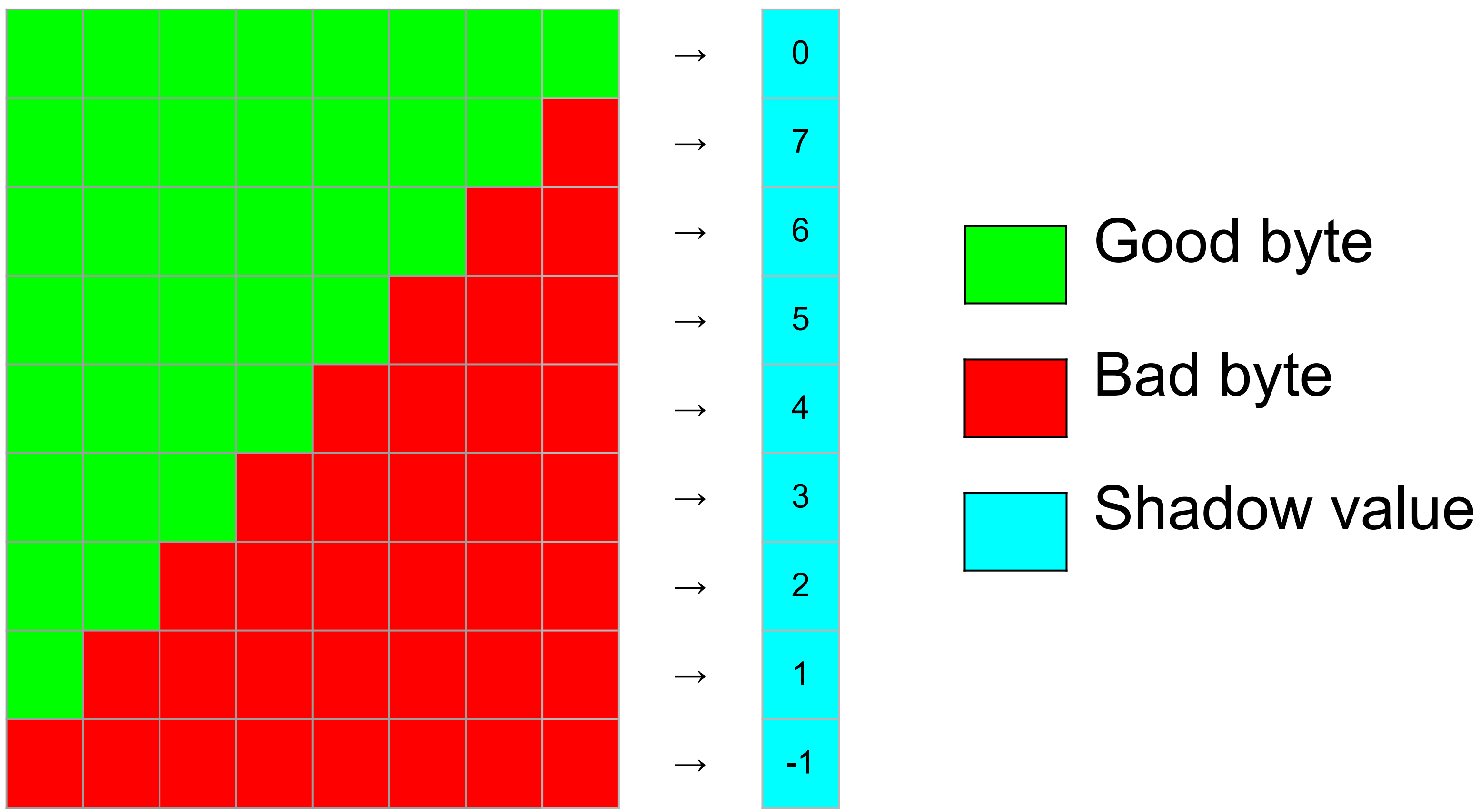
CONFIG_KASAN

```
BUG: KASAN: use-after-free in n_tty_receive_buf_common at addr ffff88006555dcb0
Read of size 1 by task syz-executor/17003
Call Trace:
 [<     inline     >] n_tty_receive_buf_fast drivers/tty/n_tty.c:1575
 [<     inline     >] __receive_buf drivers/tty/n_tty.c:1613
 [<ffffffff83234cd9>] n_tty_receive_buf_common drivers/tty/n_tty.c:1711
 [<ffffffff83235303>] n_tty_receive_buf2 drivers/tty/n_tty.c:1746
...
Allocated PID = 17003:
 [<     inline     >] kmalloc include/linux/slab.h:495
 [<ffffffff83260b69>] set_selection drivers/tty/vt/selection.c:298
 [<ffffffff8327f270>] tioclinux drivers/tty/vt/vt.c:2679
 [<ffffffff8325c1ef>] vt_ioctl drivers/tty/vt/vt_ioctl.c:365
...
Freed PID = 17034:
 [<ffffffff81807813>] kfree mm/slab.c:3837
 [<ffffffff83260b89>] set_selection drivers/tty/vt/selection.c:304
 [<ffffffff8327f270>] tioclinux drivers/tty/vt/vt.c:2679
 [<ffffffff8325c1ef>] vt_ioctl drivers/tty/vt/vt_ioctl.c:365
...
```

Shadow Byte

Any aligned 8 bytes may have 9 states: N good bytes and 8 - N bad (0 <= N <= 8).

State of such 8 aligned bytes is encoded in 1 "shadow" byte.



Shadow Memory

1/8 of address space is reserved for shadow. x86_64:

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
hole caused by [48:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffbffffffffff (=44 bits) kasan shadow memory (16TB)
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffef00000000 - ffffffffefffffff (=64 GB) EFI region mapping space
... unused hole ...
ffffffff80000000 - ffffffff9fffffff (=512 MB)  kernel text mapping, from phys 0
ffffffffa0000000 - ffffffffff5fffff (=1526 MB) module mapping space
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

Shadow Mapping

Shadow = Addr >> 3 + 0xDFFFFC0000000000

Compiler Instrumentation

```
// 8-byte memory access
*a = ...
```

```
char *shadow = (a >> 3) + Offset;
if (*shadow)
   ReportError(a);
*a = ...; // original access
```

Compiler Instrumentation (2)

```
// 1,2,4-byte memory access
*a = ...
```

```
char *shadow = (a >> 3) + Offset;
if (*shadow && *shadow < (a & 7) + N)
   ReportError(a);
*a = ...; // original access
```

Stack Instrumentation

```
void foo() {
    char a[328];



     <------------- CODE ------------->

}
```

Stack Instrumentation (2)

```
void foo() {
    char rz1[32]; // stack redzone, 32-byte aligned
    char a[328];
    char rz2[24];
    char rz3[32];



     <------------- CODE ------------->



}
```

Stack Instrumentation (3)

```
void foo() {
    char rz1[32]; // stack redzone, 32-byte aligned
    char a[328];
    char rz2[24];
    char rz3[32];

    int *shadow = (&rz1 >> 3) + kOffset; // poison redzones
    shadow[0] = 0xffffffff; // poison rz1
    shadow[11] = 0xffffff00; // poison rz2
    shadow[12] = 0xffffffff; // poison rz3

     <------------- CODE ------------->

    shadow[0] = shadow[11] = shadow[12] = 0; // unpoison redzones
}
```

Globals Instrumentation

```
// A global variable.
int a;
```

```
struct {
   int    original;
   char   redzone[60];
} a; // 32-aligned
```

Run-time Module

- maps shadow memory
- adds redzones around slab objects
- poisons/unpoisons shadow on kfree/kmalloc
- ensures delayed reuse of slab objects
- poisons global redzones on startup
- collects stack traces for kmalloc/kfree
- prints error reports

Slab Redzone

16 - 2048 bytes

```
struct kasan_alloc_meta {
    u32                     alloc_pid;
    depot_stack_handle_t    alloc_stack;
    u32                     free_pid;
    depot_stack_handle_t    free_stack;
    char                    pad[0-2032];
};
```

Stack depot (lib/stackdepot.c): Huge concurrent hashmap: stack trace -> u32.

Quarantine

Quarantine is FIFO delayed object reuse queue.
Higher chances of catching UAF.

Global but with per-CPU caches for scalability.

Max size is 1/32 of RAM.

KMSAN

uses of uninitialized memory

Uses of Uninit Memory

Not reads/copies of uninit memory.

Not uses of not-stored-to memory.

Reading/copying is OK.

Calculations are OK if result is discarded/unused.

What is a Use?

- Value affects control flow
- Value is dereferenced
- Value goes to user-space/network

What is Uninit Memory?

Uninitialized-ness comes from:

- kmalloc
- stack variables

Initialized-ness comes from:

- constants
- user-space/network

Initialized-ness is propagated/transformed by:

- loads
- stores
- computations

Shadow Memory

Bit-to-bit shadow.

1 bit of kernel memory -> 1 bit of shadow.

"0" - initialized, "1" - not initialized.

KASAN reserves 1/8 of address space for shadow. Does not work for KMSAN.

Per-page shadow:

```
 struct page {
+  void *shadow;
```

Implementation

- Runtime Part
  - kmalloc() poisons shadow.
  - memcpy() copies shadow.
  - copy_to_user() checks shadow of the kernel region.
- Compiler Part
  - Poison local vars on func entry.
  - Propagate/transform shadow on loads/stores/computations.
  - Check shadow on uses.

Shadow Propagation

A = B: A' = B'

A = B << C: A' = B' << C

A = B + C: A' = B' | C' (approx.)

A = B & C: A' = (B' & C') | (B & C') | (B' & C)

Chained Origins Example

```
int main(int argc, char** argv) {
  int* a = new int[10];
  a[5] = 0;
  volatile int b = a[argc];
  if (b)
    printf("xx\n");
  return 0;
}
```

```
WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x7f7893912f0b in main umr.c:7

  Uninitialized value was stored to memory at
    #1 0x7f7893912ecd in main umr.c:6

  Uninitialized value was created by a heap allocation
    #0 0x7f7893901cbd in operator new msan_new_delete.cc:44
    #1 0x7f7893912e06 in main umr.c:4
```

Clang!!!

We need clang for instrumentation.

We need CLANG

- KMSAN
- CFI/SafeStack
- shake out latent bugs
- warnings

KTSAN

data races

Data Race

Two threads access the same variable concurrently, and at least one of the accesses is a write.

Undefined behavior according to C standard.

Non-deterministic and very hard to debug.

CONFIG_KTSAN

```
ThreadSanitizer: data-race in do_mmap_pgoff
Read at 0xffff8800bb857e30 of size 4 by thread 1471 on CPU 0:
 [<ffffffff8121e770>] do_mmap_pgoff+0x4f0/0x590 mm/mmap.c:1341
 [<ffffffff811f8a6a>] vm_mmap_pgoff+0xaa/0xe0 mm/util.c:297
 [<ffffffff811f8b0c>] vm_mmap+0x6c/0x90 mm/util.c:315
...
Previous write at 0xffff8800bb857e30 of size 4 by thread 1468 on CPU 8:
 [<     inline     >] do_remount fs/namespace.c:2215
 [<ffffffff8129a157>] do_mount+0x637/0x1450 fs/namespace.c:2716
 [<     inline     >] SYSC_mount fs/namespace.c:2915
...
Mutexes locked by thread 1471:
Mutex 228939 is locked here:
 [<ffffffff81edf7c2>] mutex_lock_interruptible+0x62/0xa0
kernel/locking/mutex.c:805
 [<ffffffff8126bd0f>] prepare_bprm_creds+0x4f/0xb0 fs/exec.c:1172
 [<ffffffff8126beaf>] do_execveat_common.isra.36+0x13f/0xb40 fs/exec.c:1517
...
Mutexes locked by thread 1468:
Mutex 119619 is locked here:
 [<ffffffff81ee0d45>] down_write+0x65/0x80 kernel/locking/rwsem.c:62
 [<     inline     >] do_remount fs/namespace.c:2205
 [<ffffffff81299ff1>] do_mount+0x4d1/0x1450 fs/namespace.c:2716
...
```

Implementation

Similar to KASAN:

- compiler intercepts all memory accesses (function calls into runtime)
- shadow memory to store meta information (though, 4x and per-page)

Run-time library implementation is significantly more complex.

Run-time Library

Handles:

- thread management (creation, start, stop, exit)
- all synchronization primitives
- memory allocation/deallocation
- memory accesses
- collects stack traces
- prints reports on data races

Shadow Memory

8 aligned kernel bytes -> 4 8-byte shadow slots (32 bytes).

Every 8-byte slot describes 1 previous memory access:

- thread id (12 bits)
- logical timestamp (42 bits)
- size (2 bits)
- offset (3 bits)
- read/write (1 bit)
- atomic/non-atomic (1 bit)

Memory Access Handling

- check previous memory access for a potential data race
- store current access in a shadow cell

Race if the two memory access:

- in different threads
- overlap
- at least one is a write
- not synchronized

Race detection algorithm

- Happens-before race detector
- Synchronization establishes happens-before
- Accesses ordered by happens-before are synchronized
- Accesses not ordered by happens-before are concurrent

Vector Clocks

- Based on Vector Clocks
- Each thread and synchronization primitive has an associated Vector Clock
- Vector Clocks are updated on sync operations
- Vector Clocks allow to say when 2 memory accesses are synchronized or not:

```
synchronized = thr->clock[other_tid] > other_timestamp
```

"Benign" Data Races

Unmarked concurrent accesses that are not considered bugs by developers.

Make KTSAN infeasible.

Say **No** to "Benign" Data Races

- Proving benignness is time consuming and impossible
- Allows automatic data race bug detection
- Makes code better documented

Proving Benignness

```
*p = (*p & 0xfffff) | v;
```

Option 1:

```
0: mov (%rdi),%rax
3: and $0xfffff,%eax
8: or %rax,%rsi
B: mov %rsi,(%rdi)
```

Option 2:

```
0: andq $0xfffff,(%rdi)
7: or %rsi,(%rdi)
```

This should be atomic, right?

```
void foo(int *p, int v)
{
    // some irrelevant code
    *p = v;
    // some irrelevant code
}
```

This should be atomic, right?

```
void foo(int *p, int v)
{
    // some irrelevant code
    *p = v;
    // some irrelevant code
}

void bar(int *p, int f)
{
    int tmp = *p & MASK;
    tmp |= f;
    foo(p, tmp);
}
```

This should be atomic, right?

```
void foo(int *p, int v)
{
    // some irrelevant code
    *p = v;
    // some irrelevant code
}

void bar(int *p, int f)
{
    int tmp = *p & MASK;
    tmp |= f;
    foo(p, tmp);
}

*p = (*p & MASK) | f;
```

This should be atomic, right? Maybe

```
void foo(int *p, int v)
{
    // some irrelevant code
    *p = v;
    // some irrelevant code
}

void bar(int *p, int f)
{
    int tmp = *p & MASK;
    tmp |= f;
    foo(p, tmp);
}

*p = (*p & MASK) | f;

0: andq $0xffffff,(%rdi)
7: or %rsi,(%rdi)
```

Based on Real Bug

```
--- a/fs/namespace.c
+++ b/fs/namespace.c
@@ -2212,7 +2212,7 @@ static int do_remount(struct path
*path, int flags, int mnt_flags,
                 lock_mount_hash();
                 mnt_flags |= mnt->mnt.mnt_flags &
                                 ~MNT_USER_SETTABLE_MASK;
-                mnt->mnt.mnt_flags = mnt_flags;
+                WRITE_ONCE(mnt->mnt.mnt_flags, mnt_flags);
                 touch_mnt_namespace(mnt->mnt_ns);
                 unlock_mount_hash();
```

Temporary exposes mount without MNT_NOSUID, MNT_NOEXEC, MNT_READONLY flags.

Fragile

- Changing local computations can break such code
- Changing MASK from 0xfe to 0xff can break such code
- New compiler can break such code
- LTO can break such code

Optimizations vs Atomic Accesses

Want fast code!

Impossible to draw a line between what should be optimized and what should be atomic with unmarked accesses.

These requirements are in direct conflict.

Tooling

Dynamic/static race detection

```
--- a/drivers/tty/tty_buffer.c
+++ b/drivers/tty/tty_buffer.c
          n->flags = flags;
          buf->tail = n;
-         b->commit = b->used;
+         smp_store_release(&b->commit, b->used);
```

```
--- a/mm/vmalloc.c
+++ b/mm/vmalloc.c
-               smp_rmb();
                if (v->flags & VM_UNINITIALIZED)
                        return;
+               smp_rmb();
```

Code Documentation

Synchronization is an important aspect and it must be visible in code:

```
--- a/drivers/tty/tty_buffer.c
+++ b/drivers/tty/tty_buffer.c
@@ -467,7 +467,7 @@ static void flush_to_ldisc(struct
work_struct *work)

        tty = port->itty;

        if (tty == NULL)
                return;
        disc = tty_ldisc_ref(tty);
```

Code Documentation

Synchronization is an important aspect and it must be visible in code:

```
--- a/drivers/tty/tty_buffer.c
+++ b/drivers/tty/tty_buffer.c
@@ -467,7 +467,7 @@ static void flush_to_ldisc(struct
work_struct *work)

-       tty = port->itty;
+       tty = READ_ONCE(port->itty);
        if (tty == NULL)
                return;
        disc = tty_ldisc_ref(tty);
```

What is lifetime of tty?

Can Have Memory Model

Data races are undefinable.

Would greatly simplify Paul's work on the memory model.

No Overhaul

No need to change everything at once.

Just don't push back on patches written by other people.

Can be KTSAN driven.

Pain Points

Variety of oops messages

"BUG:", "WARNING:", "INFO:", Unable to handle kernel paging request", "general protection fault:", "Kernel panic", "kernel BUG", "Kernel BUG", "divide error:", "invalid opcode:", "unreferenced object", "UBSAN:", ...

- Difficult to even find/grep.
- Not possible to detect begin/end of oops.
- Ad-hoc formats:
  - printk(KERN_ALERT "BUG: Bad rss-counter state mm:%p idx:%d val:%ld\n");
- Oops that are not oops:
  - [INFO: suspicious RCU usage.]
  - INFO: lockdep is turned off.
- No identification (the same or new?)

Ideally:

- strstr() to detect all oopses
- strstr() to find begin/end of oops
- provide unique oops "ID": SEVERITY: bug-type in function

Other Pain Points

- Reported bugs are not always fixed
- WARNINGs on invalid syscall inputs
- User-space address space layout changes (0x2axx->0x7fxx->0x55xx)

Thanks!

Q&A

Dmitry Vyukov, dvyukov@